

Semestrální práce - KIV/PPR

Raytracing

AUTOR:

Jakub Křížanovský

A24N0097P

krizanoj@students.zcu.cz

Zadání

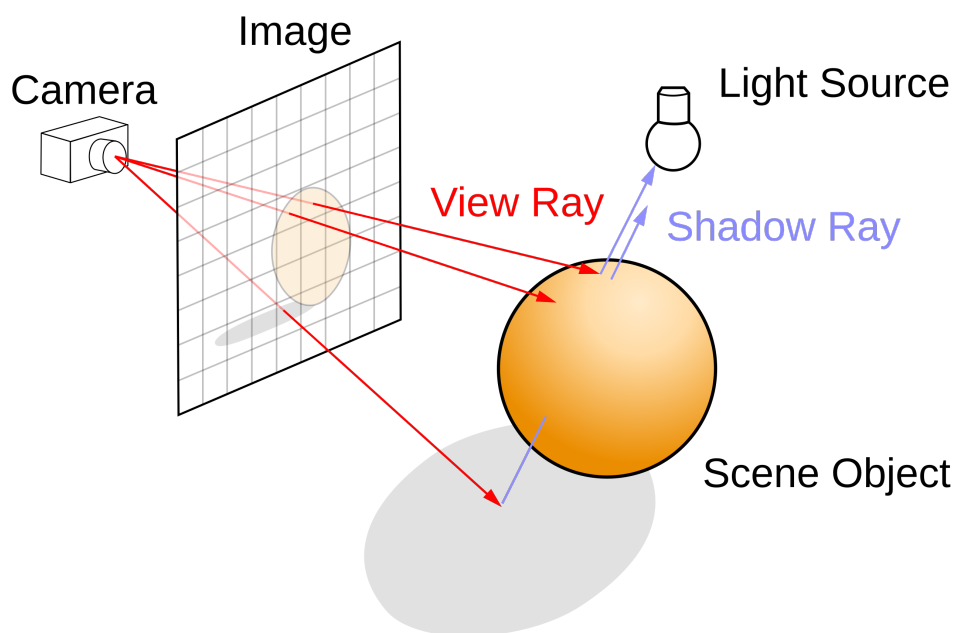
Cílem práce je návrh a implementace systému pro vykreslování jednoduchých scén tvořených geometrickými primitivami (koulemi) pomocí techniky *raytracing*. Důrazem této práce je využití paralelizačních technik pro urychlení výpočtu výsledného obrazu a jejich následné srovnání.

V rámci práce budou implementovány a srovnány tři způsoby paralelizace při výpočtu. Základní způsob bude sekvenční, čili bez paralelizace a bude sloužit jako referenční model pro ověření správnosti algoritmu a měření zrychlení. Druhý způsob bude využití vícevláknového zpracování v kombinaci s vektorovými instrukcemi (*SIMD* - single instruction, multiple data). Třetí způsob bude realizace výpočtu paralelně na grafické kartě.

Raytracing

Raytracing (sledování paprsků) je metoda renderování 3D počítačové grafiky. V reálném světě jsou zdroji paprsků světelné zdroje, od kterých se šíří různými směry, odráží a lámou se při kolizi s povrchy. Naprostá většina těchto paprsků ale nezasáhne oko pozorovatele a tak by tento výpočet byl extrémně neefektivní.

Na rozdíl od běžného života je tedy v rámci tohoto algoritmu směr paprsků obrácený. Z bodu pozorování (kamery) je skrze každý pixel budoucího obrazu vyslán jeden paprsek směrem do scény. Pokud tento paprsek zasáhne objekt, vypočítá se v bodě střetu barva na základě vlastností materiálu a pozice světla. Od tohoto bodu se mohou odrážet další (sekundární) paprsky, které umožňují věrně vykreslit stíny, odrazy nebo lom světla v průhledných materiálech.



Princip raytracingu [1]

Návrh

Technologie

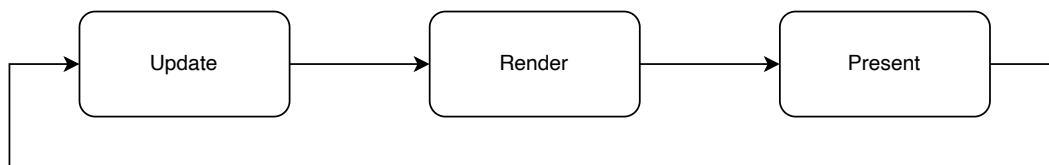
Pro implementaci systému jsem zvolil následující technologie:

- programovací jazyk C++
- *Vulkan* - API pro přístup k grafice a výpočtům na GPU
- *GLSL (OpenGL Shading Language)* - programovací jazyk pro psaní shaderů
- *ARM Neon* - sada instrukcí SIMD pro architekturu ARM
- *oneTBB (Threading Building Blocks)* - knihovna pro vícevláknové zpracování úloh na CPU
- *GLM (OpenGL Mathematics)* - knihovna pro matematické operace
- *GLFW* - multiplatformní knihovna pro práci s okny
- *nlohmann/json* - knihovna pro serializaci a deserializaci souborů ve formátu *json*

Základní aplikační smyčka

V rámci implementace je využita standardní aplikační smyčka. Celý proces je rozdělen do tří logických fází, které se cyklicky opakují. Jedná se o následující:

- **Update:** Tato fáze slouží k přípravě dat před samotným výpočtem. Po vzoru běžných simulací či herních enginů je v této fázi místem pro řešení pohybu objektů - aktualizaci jejich polohy. V rámci této práce je využita pouze pro pohyb kamery kolem středového bodu. Pro případné další vylepšení je ale myšlena i pro využití na pohyb těles v rámci scény.
- **Render:** Jádro aplikace, ve kterém probíhá samotné renderování obrazu metodou raytracingu. V této fázi se aplikují zvolené paralelizační techniky (sekvenční běh, SIMD nebo GPU výpočet). Algoritmus pro každý pixel scény vypočítá výslednou barvu tohoto pixelu.
- **Present:** Závěrečná fáze slouží k vizualizaci vypočtených dat. Vygenerovaná data s barvami pixelů (buffer / obrázek) jsou zobrazena uživateli na monitoru. Po dokončení této fáze začíná renderování dalšího snímku a program přechází zpět k aktualizaci stavu scény.



Aplikační smyčka

Architektura systému

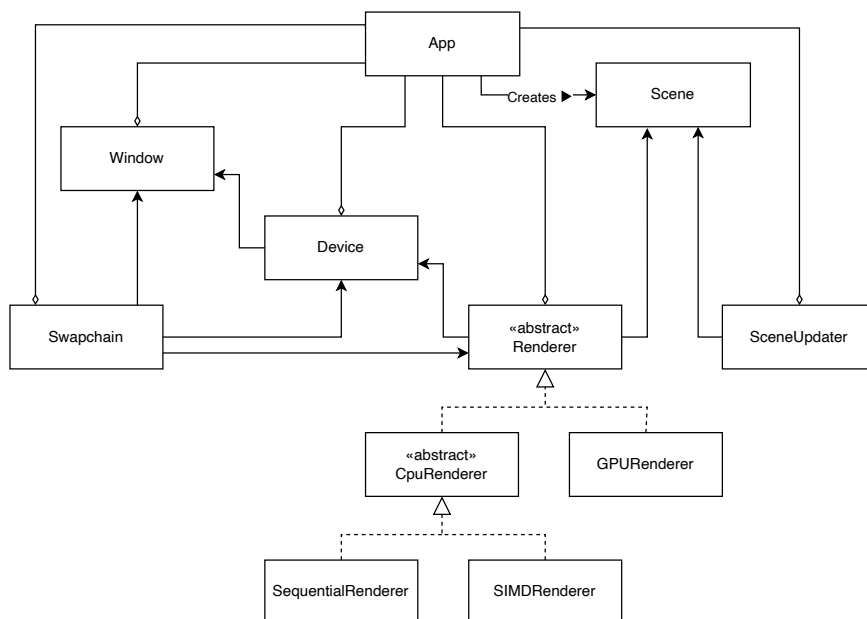
Při návrhu architektury systému bylo nutné splnit jednu hlavní podmínku - modularitu. Celý systém musel být navržen tak, aby jeho části šly transparentně vyměnit na základě zvoleného způsobu paralelizace. Také bylo nutné oddělit infrastrukturu aplikace, která je stejná pro všechny typy, od výpočetní logiky renderování. K dosažení těchto cílů jsem se rozhodl využít principů dědičnosti a polymorfismu, což považuji za nejčistší řešení daného problému.

Základem abstrakce výpočetní logiky je abstraktní třída *Renderer*. Ta definuje jednotné rozhraní, skrze které pak ostatní části aplikace interagují s procesem renderování scény. Konkrétní implementace abstraktní třídy *Renderer* jsou pak tři - třída *SequentialRenderer* s implementací sekvenční verze výpočtu, *SIMDRenderer* s implementací výpočtu pomocí vláken a vektorových instrukcí a *GPURenderer* implementující výpočet na GPU. Třídy *SequentialRenderer* a *SIMDRenderer* mají navíc ještě jednu společnou vlastnost - data jsou při renderování zapisována do pracovního bufferu, který je následně překopírován do obrázku swapchainu. Pro zachování principů objektově orientovaného programování je jejich společná logika ještě oddělena do abstraktní třídy *CPURenderer*, od které obě dědí.

Další částí aplikace jsou třídy tvořící infrastrukturu - *Window*, *Device* a *Swapchain*. Tyto třídy slouží k zapouzdření logiky komunikace s hardwarem pomocí *Vulkan API* a s operačním systémem.

Třída *Window* má na starost vytvoření a správu okna aplikace, k čemuž využívá knihovnu *GLFW*. Dále je zde třída *Device*, která zapouzdřuje logické grafické zařízení (*VkDevice*). Třída je zodpovědná za inicializaci grafického rozhraní, správu front příkazů a alokaci paměti pro zdroje, jako jsou buffery či obrázky. Poslední z těchto tříd je třída *Swapchain*, která spravuje řetězec obrazových bufferů a má na starost prezentaci výsledných obrázků uživateli (vykreslení do okna).

Středovým bodem řídícím celou aplikaci je třída *App*. Tato třída má na starost inicializaci a správu všech komponent, přičemž na základě typu paralelizace vytváří odpovídající instanci rendereru. Dále vytváří a spravuje instanci třídy *Scene*, která tvoří datový model pro výpočet. Poslední zásadní zodpovědností je správa základní aplikační smyčky, která řídí chod celé aplikace.



Architektura systému

Paralelizace

Vzhledem k povaze algoritmu raytracingu, kde výpočet barvy jednoho pixelu nijak neovlivňuje barvu pixelů sousedních, je paralelizace realizována na úrovni jednotlivých pixelů obrazu. Konkrétní strategie zpracovávání těchto pixelů se pak liší na základě zvoleného typu paralelizace.

SIMD

Pro tento typ paralelizace jsou využita CPU vlákna v kombinaci s vektorovými instrukcemi (konkrétně *ARM Neon*). Pro použití vektorových instrukcí je nevhodné mapovat jeden 3D vektor přímo na jeden SIMD vektor, neboť by nevyužitá čtvrtá složka registru vedla k neefektivitě.

Vhodnější reprezentací je zpracování čtyř pixelů současně. Jejich složky *x*, *y* a *z* jsou reorganizovány do tří samostatných *SIMD* vektorů (např. jeden *SIMD* vektor obsahuje složky *x* všech čtyř paprsků). Celý obraz je dále ještě rozdělen na řádky, které distribuovány mezi jednotlivá vlákna.

GPU

Při výpočtu na grafické kartě je využito masivně paralelní architektury, kde je výpočet rozdělen na jednotlivé pracovní položky odpovídající pixelům výsledného obrazu. Každý pixel je zpracováván samostatnou invokací compute shaderu, což umožňuje současné spuštění tisíců vláken na výpočetních jednotkách GPU.

Datové struktury

Nedílnou součástí návrhu systému je definice datových struktur. Datové struktury jsem rozdělil na dvě skupiny. První skupinou jsou datové struktury popisující scénu, které tvoří datový model. Tyto struktury jsou společné pro všechny typy paralelizace a jsou vstupem algoritmu. Druhou částí jsou struktury sloužící pro výpočet, které se liší pro konkrétní typy paralelizace.

Reprezentace scény

Základním prvkem je struktura *Scene*, která slouží jako kontejner pro všechny entity. Struktura agreguje:

- **Kameru** (struktura *Camera*) - Uchovává parametry pozorovatele včetně pozice, směru pohledu a velikosti zorného pole (*fov*). Pro účely animace obsahuje také data pro orbitální pohyb - střed orbity a rychlost.
- **Osvětlení** (struktura *LightData*) - Definuje osvětlení scény - směr a barvu směrového světla a ambientní světlo.
- **Objekty** (pole struktur *Sphere*) - Seznam objektů - koulí ve scéně. Každá koule je definovaná středem, poloměrem a barvou.

Scene	Camera	LightData	Sphere
Camera camera	glm::vec3 position	glm::vec3 directionalLightDirection	glm::vec3 position
LightData lightData	glm::vec3 forward	glm::vec3 directionalLightColor	float radius
Sphere[] spheres	float fov	glm::vec3 ambientLightColor	glm::vec3 color
	glm::vec3 focusPoint		
	float orbitVelocity		

Datové struktury - reprezentace scény

Datové struktury pro výpočet

Pro samotný proces raytracingu jsou definovány pomocné struktury, které se liší podle způsobu zpracování dat. Jedná se o struktury:

- **3D Vektor** - Základní prvek pro reprezentaci pozic a směrů v prostoru. Kromě toho je také využíván pro reprezentaci barev (*RGB* složky).
- **Paprsek** (*Ray*) - Struktura definovaná počátkem (*origin*) a směrovým vektorem (*direction*).
- **Zásah paprsku** (*RaycastHit*) - Bod průsečíku paprsku s objektem - obsahuje pozici průsečíku, normálu povrchu objektu v tomto bodě a vzdálenost od počátku paprsku k bodu.

Pro sekvenční a *GPU* varianty používají tyto datové struktury 3D vektory z knihovny *glm* (v případě *GLSL* compute shaderu jsou to nativní typy tohoto jazyka) a jednoduché datové typy s plovoucí řádovou čárkou (*float*).

Pro *SIMD* variantu výpočtu jsou pak nadefinované vlastní specializované vektory (*Vec3_x4*), které využívají vektory pro *Neon* instrukce typu *float32x4_t*. Je tak umožněno zpracovávat čtyři nezávislé paprsky (v rámci struktur *Ray_x4* a *RaycastHit_x4*) v jediné instrukci.

glm::vec3	Ray	RaycastHit	Vec3_x4	Ray_x4	RaycastHit_x4
float x	glm::vec3 origin	glm::vec3 position	float32x4_t x	Vec3_x4 origin	Vec3_x4 position
float y	glm::vec3 direction	glm::vec3 normal	float32x4_t y	Vec3_x4 direction	Vec3_x4 normal
float z		float distance	float32x4_t z		float32x4_t distance

Datové struktury - sekvenční a GPU

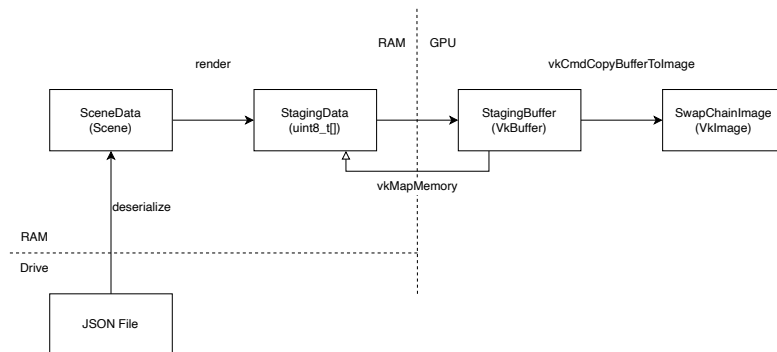
Datové struktury - SIMD

Předávání dat

Poslední důležitou součástí návrhu je proces předávání dat mezi jednotlivými částmi aplikace. Tento proces začíná načtením scény z disku a končí zobrazením výsledného snímku swapchainu, přičemž se liší podle toho, zda výpočet probíhá na CPU nebo GPU.

CPU renderery

V případě sekvenčního a *SIMD* výpočtu probíhá renderování přímo do pomocného pole v RAM (*StagingData*). Aby bylo možné tato data zobrazit pomocí Vulkan API, musí být přenesena na grafickou kartu. To je řešeno mapováním GPU bufferu (*StagingBuffer*) do paměťového prostoru CPU příkazem *vkMapMemory*. Následně je vyvolán příkaz *vkCmdCopyBufferToImage*, který data z bufferu překopíruje do finálního obrazu swapchainu pro zobrazení.



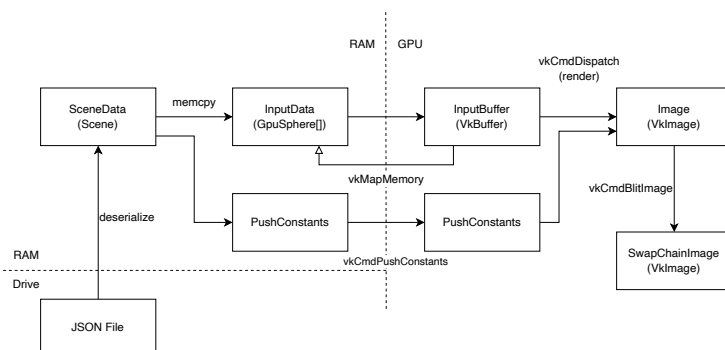
Předávání dat - CPU renderery

GPU renderer

V případě GPU varianty výpočtu je situace trochu odlišná, vzhledem k tomu, že vstupní data je potřeba přímo předat GPU. Data jsou opět načtena ze souboru do struktury *Scene*. Dále jsou pak předávána GPU dvěma různými způsoby.

- Objekty scény jsou předávány přes input buffer pomocí mapování paměti. Nejprve je ale nutné transformovat je na GPU variantu (struktura *GPUSphere*) pro správné zarovnání vektorů podle formátu *std140*.
- Zbývající data jsou na GPU předávána pomocí push konstant. Jedná se o malá data (jednotky bytů) ze kterých se některá mohou měnit každý snímek (například pozice kamery), proto je využití mechanismu push konstant vhodné. Mezi tyto data patří data kamery, osvětlení scény, počet objektů a velikost textury pro výstup.

Dále už je příkazem *vkCmdDispatch* spuštěn compute shader, který čte data z obou těchto vstupů a výsledek zapisuje do textury. Finální obraz je pak pouze překopírován do swapchain obrázku pro prezentaci uživateli.



Předávání dat - GPU renderer

Princip výpočtu obrazu

Výpočet barvy každého pixelu scény probíhá na základě jednotného modelu, který se skládá z následujících kroků:

- **Perspektivní kamera** - Pro každý pixel výsledného obrazu je vygenerován paprsek, jehož směr je ovlivněn zorným polem (FOV). Tento model simuluje přirozené lidské vidění, kde se objekty se zvyšující se vzdáleností od pozorovatele zdají menší.
- **Průsečík paprsku s koulí** - Detekce kolizí probíhá pomocí analytického řešení kvadratické rovnice, která definuje průnik přímky (paprsku) a koule.
- **Phongův osvětlovací model** - Pro určení výsledné barvy povrchu v bodě zásahu je implementován osvětlovací model, který se skládá ze tří složek:
 - **Ambientní složka** - Reprezentuje rozptýlené světlo v prostředí, které zajišťuje, že i plochy v úplném stínu nejsou zcela černé.
 - **Difúzní složka** - Simuluje rozptyl světla na matném povrchu. Její intenzita závisí na úhlu mezi normálou povrchu v bodě zásahu a směrem k dopadajícímu (směrovému) světlu.
 - **Spekulární složka** - Vytváří světelné odlesky na lesklých materiálech, přičemž bere v úvahu směr odrazu světla vůči směru paprsku
- **Odrazy (rekurze)** - V bodě zásahu je na základě normály povrchu vygenerován sekundární paprsek, který vypočítává barvu okolního prostředí. Tato barva je následně zahrnuta do výpočtu barvy povrchu objektu, což umožňuje věrné zobrazení odrazů světla na lesklých objektech. Maximální počet odrazů je omezen konstantou, což umožňuje algoritmu vždy v konečné době skončit.

Algoritmus

Problém raytracingu přirozeně vede na využití rekurzivního algoritmu. Při výpočtu průsečíku paprsku s objektem lze rekurzivně zavolat opět stejnou metodu pro zjištění barvy odraženého světla, kterou je možné zahrnout do výpočtu barvy povrchu. Vzhledem k vlastnostem rekurzivních algoritmů, jsem se ale v první fázi rozhodl pro implementaci jiného, nerekurzivního řešení.

V rámci tohoto řešení jsem rozdělil výpočet barvy pixelu na dvě iterativní fáze - dopředný chod a zpětný chod. V první fázi výpočtu jsou počítány průsečíky paprsků s objekty ve scéně a paprsky vždy pro následující iteraci. Data všech paprsků, odrazů a barvy objektů jsou v rámci toho ukládána do pomocného pole. V druhé fázi pak jde algoritmus zpět od posledního odrazu a postupně vypočítává výsledné světlo na základě dat v pomocném poli.

Algoritmus je možné popsat následujícím pseudokódem:

```
funkce raycast(paprsek):  
    // fáze 1 - dopředný chod  
    paprsky[0] = paprsek  
    for i od 0 do MAX_ODRAZŮ:  
        najdi nejbližší průsečík paprsku[i] se všemi koulemi ve scéně  
        ulož data o zásahu (pozice, normála, barva objektu)  
        vypočítej odražený paprsek[i+1] pro další iteraci  
  
    // fáze 2 - zpětný chod  
    výsledná_barva = ČERNÁ (0, 0, 0)  
    for i od MAX_ODRAZŮ do 0:  
        if(zásah[i].zasáhnul):  
            výsledná_barva *= intenzita_odrazu * barva_povrchu  
            výsledná_barva += osvětlení na základě modelu  
  
    return výsledná_barva
```

Měření výsledků

Pro měření výsledků jsem použil postup přes měření času pomocí knihovny *chrono*. V základní smyčce aplikace (v třídě *App*) jsem přidal dvě zaznamenání času - jedno před vykreslením snímku a jedno po vykreslení snímku. Doba pro výpočet snímku je pak vypočtena jako rozdíl mezi těmito časy. Pro eliminaci náhodných výkyvů je pak vypočítán aritmetický průměr za celou dobu běhu testovací sekvence.

Pro měření jsem použil tři scény s rozdílnou výpočetní náročností (jsou součástí repozitáře ve složce *scenes*) - *default.json* (3 objekty), *random20.json* (20 objektů) a *random100.json* (100 objektů).

Všechna měření provádím na svém Macbooku Air s čipem M1 (8 jader CPU + integrovaná GPU). Výsledky měření pro jednotlivé renderery a scény jsou uvedeny v následující tabulce (data všech tabulek jsou v milisekundách):

	Sequential	SIMD	GPU
Default	1852	1192	15,58
Random20	8660	6303	15,89
Random100	38236	23969	51,48

Výsledky původního algoritmu

Metriky

Na základě hodnot z tabulky můžeme pro typy paralelizace SIMD a GPU vypočítat metriky urychlení a Karp-Flattovu metriku. Metriky jsou počítány z výsledků pro 100 objektů.

Urychlení výpočtu těchto dvou variant lze vypočítat následovně:

$$S_{SIMD} = \frac{T_{seq}}{T_{SIMD}} = \frac{38236}{23969} \simeq 1,596$$

$$S_{GPU} = \frac{T_{seq}}{T_{GPU}} = \frac{38236}{51,48} \simeq 742,735$$

Pro výpočet Karp-Flattovy metriky je nejprve určit počet procesorů P . Tento počet jsem pro SIMD variantu určil jako:

$$p_{SIMD} = \text{počet jader CPU} \times \text{počet složek SIMD vektoru}$$

$$p_{SIMD} = 8 \times 4 = 32$$

Pro GPU variantu jsem ho na základě vlastností integrované grafické karty čipu M1 [2] určil jako:

$$p_{GPU} = \text{počet jader GPU} \times \text{počet EU / jádro} \times \text{počet ALU / EU}$$

$$p_{GPU} = 8 \times 16 \times 8 = 1024$$

Na základě těchto hodnot pak můžeme dopočítat Karp-Flattovu metriku následujícím způsobem:

$$e_{SIMD} = \frac{\frac{1}{S_{SIMD}} - \frac{1}{p_{SIMD}}}{1 - \frac{1}{p_{SIMD}}} = \frac{\frac{1}{1,596} - \frac{1}{32}}{1 - \frac{1}{32}} \simeq 0,680$$

$$e_{GPU} = \frac{\frac{1}{S_{GPU}} - \frac{1}{p_{GPU}}}{1 - \frac{1}{p_{GPU}}} = \frac{\frac{1}{742,735} - \frac{1}{1024}}{1 - \frac{1}{1024}} \simeq 0,0023$$

Rozbor a optimalizace

Z výsledků měření vychází, že paralelizace na GPU funguje pro tento problém velmi dobře. Bohužel ale urychlení pro SIMD renderer (paralelizaci pomocí vláken a vektorových instrukcí) vychází dost malé. Vzhledem k osmi jádrům procesoru a čtyřem zpracovávaným instrukcím naráz by bylo možné očekávat teoretické urychlení až 32. Prakticky této hodnoty nebude možné dosáhnout, ale urychlení okolo hodnoty 1,6 má k teoretickému výsledku hodně daleko.

Analýza rychlosti SIMD výpočtu

Pro bližší identifikaci problému jsem provedl další měření - pouze s použitím SIMD instrukcí a pouze s použitím vícevláknového zpracování (pomocí *TBB*). Výsledky těchto měření jsou doplněny v následující tabulce:

	Sequential	SIMD + TBB	SIMD only	TBB only
Default	1852	1192	5918	409
Random20	8660	6303	29896	1930
Random100	38236	23969	116026	8987

Výsledky původního algoritmu - rozdělení na SIMD a TBB

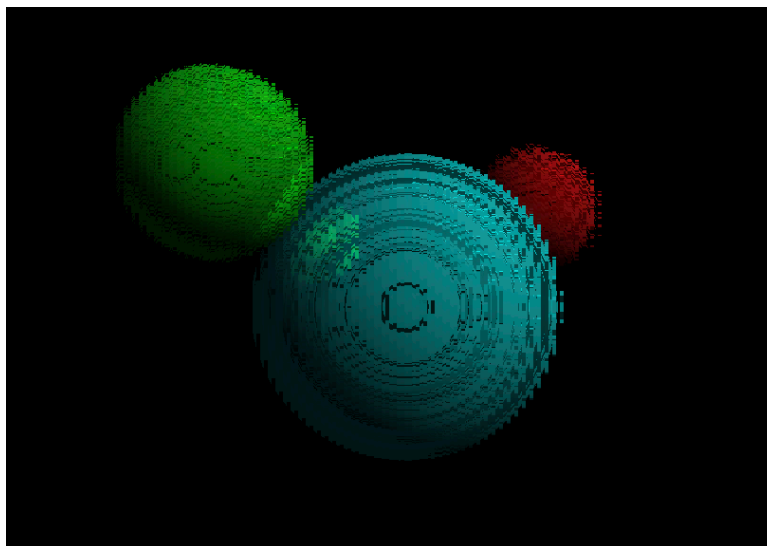
Z výsledků měření je patrné, že problém není v vícevláknovém zpracování. Urychlení výpočtu pouze za použití vláken je kolem hodnoty 4,25. To by relativně odpovídalo, vzhledem k tomu, že můj procesor Apple M1 obsahuje čtyři výkonná (*Firestorm*) a čtyři úsporná jádra (*Icestorm*). Sekvenční výpočet tak může běžet na výkonném jádře a osminásobného zrychlení nelze dosáhnout.

Z výsledků měření je dále vidět jasný problém u vektorového výpočtu, který je oproti referenčnímu sekvenčnímu výpočtu přibližně třikrát pomalejší. V další části práce se proto pokusím tyto hodnoty vylepšit prostřednictvím optimalizace algoritmu.

Snížení přesnosti SIMD výpočtu

Jedním z pokusů o vylepšení vlastností *SIMD* rendereru bylo snížení přesnosti operací na polovinu. V rámci *Neon* operací tedy budeme místo *float16x4_t* typů pracovat s typy *float8x8_t* a s instrukcemi pracujícími s těmito typy. Tím zdvojnásobíme počet pixelů, které jsme schopni zpracovávat ze čtyř na osm.

Podle očekávání tento přístup zdvojnásobil rychlost celého výpočtu. Bohužel však poloviční přesnost nestačí pro získání použitelného obrazu. Výsledek tohoto renderování je následujícím obrázkem:



Obráz po snížení přesnosti SIMD výpočtu

Úprava algoritmu

Po analýze algoritmu jsem došel k závěru, že hlavním problémem SIMD výpočtu bude takzvaný *register spilling*. V průběhu dopředného chodu ukládá algoritmus data do pomocných struktur. Jedná se převážně o data typů *Ray_x4* a *RaycastHit_x4*, která se dále skládají mimo jiné z několika *Vec3_x4*. Každá tato struktura tedy uchovává několik *Neon* registrů. Vzhledem k tomu, že počet registrů v procesoru je omezený, musí CPU při jejich nedostatku data průběžně ukládat do operační paměti a následně je znovu načítat. Právě tato režie spojená s častou manipulací s pamětí by mohla být důvodem pomalého výpočtu.

Na základě této hypotézy jsem se rozhodl algoritmus přepracovat. Namísto dvou průchodů (dopředného a zpětného) je možné výpočet realizovat pouze v rámci jednoho průchodu. Tento přístup sice mírně komplikuje průběžný výpočet příspěvku odraženého světla do výsledné barvy pixelu, ale zásadně redukuje potřebu ukládání mezivýsledků.

```
funkce raycast(paprsek):  
    výsledná_barva = ČERNÁ (0, 0, 0)  
    přínos = BÍLÁ (1, 1, 1)  
    for i od 0 do MAX_ODRAZŮ:  
        Najdi nejbližší průsečík paprsku se všemi koulemi ve scéně  
  
        if(žádný paprsek nezasáhl):  
            break;  
  
        výsledná_barva += přínos * osvětlení na základě modelu  
  
        paprsek = vypočítej odražený paprsek pro další iteraci  
        přínos *= intenzita_odrazu * barva_koule  
    return výsledná_barva
```

Namísto ukládání celých polí struktur, stačí v této verzi držet pouze aktuální hodnoty a jednou proměnnou pro akumulovaný přínos paprsku do výsledné barvy. Tím algoritmus eliminuje problém s *register spilling*. Dále navíc přidává podmínku, která předčasně ukončuje výpočet barvy, pokud žádný z paprsků nic nezasáhne.

Změna algoritmu značně vylepšila efektivitu renderování. Změnu algoritmu jsem promítnul do všech rendererů a znovu změřil nové hodnoty. Nově naměřená data jsou uvedena v následující

	Sequential	SIMD + TBB	SIMD only	TBB only	GPU
Default	550	154	725	120	15,77
Random20	2197	860	4130	511	15,69
Random100	17553	7416	36072	4162	17,62

Výsledky nového algoritmu

tabulce:

Z dat je vidět, že změna algoritmu značně vylepšila všechny typy renderování. Hlavní zásluhu za toto zrychlení odhaduji na podmínku, která umožní opustit smyčku dříve, pokud paprsek (u SIMD žádný z paprsků) nezasáhne objekt. U SIMD rendereru je ale nárůst výkonu ze všech typů nejmarkantnější (3-8x rychlejší než původní verze). U toho příčinu odhaduji právě na eliminaci *register spilling*.

Optimalizace SIMD rendereru

Stále i po přepracování algoritmu je SIMD renderer ale pomalejší než sekvenční verze. U minulé optimalizace bylo vidět, že podmínka, která dovolila opustit smyčku dříve, měla velký význam na rychlost. Provedl jsem tedy revizi kódu SIMD rendereru a našel dvě další místa kam bylo možné doplnit podobné podmínky.

Prvním místem bylo v rámci iterování přes objekty scény. Zde je počítána maska, které paprsky zasáhly daný objekt. Na základě masky pak byla nastavena data konkrétního zásahu. Zde jsem doplnil podmínku, že pokud je maska pro všechny paprsky nulová (žádný z paprsků nezasáhl objekt), výpočet pro tento objekt lze přeskočit. Tím se eliminuje velké množství operací, které by stejně byly později maskovány a zahozeny.

Druhé místo, kam bylo možné doplnit podobnou podmínku, je v rámci výpočtu průsečíku koule a paprsku. Zde je počítána kvadratická rovnice pomocí determinantu, na základě kterého je později opět maskováno. Zde jsem přidal podmínku, že pokud je determinant záporný (rovnice nemá řešení), je vrácena přímo nulová maska. Další výpočty včetně výpočetně náročných operací jako dělení a odmocnin tak mohou být přeskočeny.

Po provedení těchto změn jsem změnil nové hodnoty pro renderování pomocí vektorových instrukcí. Výsledné hodnoty jsou (společně se zbytkem s předchozích) v následující tabulce:

	Sequential	SIMD + TBB	SIMD only	TBB only	GPU
Default	550	80	375	120	15,77
Random20	2197	330	1604	511	15,69
Random100	17553	2947	14102	4162	17,62

Výsledky po optimalizacích

Z dat je vidět, že po optimalizacích již je výkonnost SIMD rendereru (bez vláken) vyšší než sekvenčního.

Finální hodnoty metrik

Finální hodnoty urychlení a Karp-Flattovy metriky jsou následující:

$$S_{SIMD} = 5,956$$

$$S_{GPU} = 996,197$$

$$e_{SIMD} = 0,206$$

$$e_{GPU} = 0,0020$$

Hlavní věcí, která je z metrik vidět, je značné vylepšení hodnot pro SIMD variantu výpočtu.

Uživatelská příručka

Sestavení a spuštění aplikace

Aplikace se sestavuje pomocí nástroje *make*. V kořenové složce adresáře je umístěn soubor *Makefile*, popisující postup sestavení. V rámci sestavování aplikace jsou také přeloženy shadery do formátu *SPIR-V*.

Sestavení aplikace se provede příkazem *make*. Aplikace lze pomocí nástroje také spustit příkazem *make run*. Ještě je k dispozici také příkaz *make clean*, pomocí kterého lze vyčistit přeložené soubory.

Pro sestavení je nutné mít správně nainstalovaný framework *Vulkan* a potřebné knihovny (*glm*, *glfw3*, *nlohmann/json*, ...). Dále je potřeba mít v souboru *.env* nastaveny cesty k *Vulkan*, k těmto knihovnám a k programu *glslc* sloužícímu pro překlad *glsl* shaderů do formátu *SPIR-V*.

Parametry

Při spouštění aplikace očekává dva parametry - typ paralelizace a cestu k souboru scény. K dispozici jsou následující typy paralelizace:

- *--sequential* - sekvenční běh (žádná paralelizace)
- *--simd* - paralelizace pomocí vláken a vektorových instrukcí
- *--gpu* - paralelizace pomocí GPU.

Jako druhý nepovinný parametr očekává aplikace cestu k souboru se scénou. Je připraveno několik ukázkových souborů s daty ve složce *scenes*. Pokud není parametr cesty nastaven, použije se soubor *scenes/default.json*.

Ukázka překladu a spuštění

Pro překlad programu a spuštění například scény *random20* za použití SIMD rendereru je nutné zavolat následující příkazy:

```
make
```

```
./raytracer.out --simd scenes/random20.json
```

Alternativně je možné využít *make* cíle *run* následujícím způsobem:

```
make run ARGS="--simd scenes/random20.json"
```

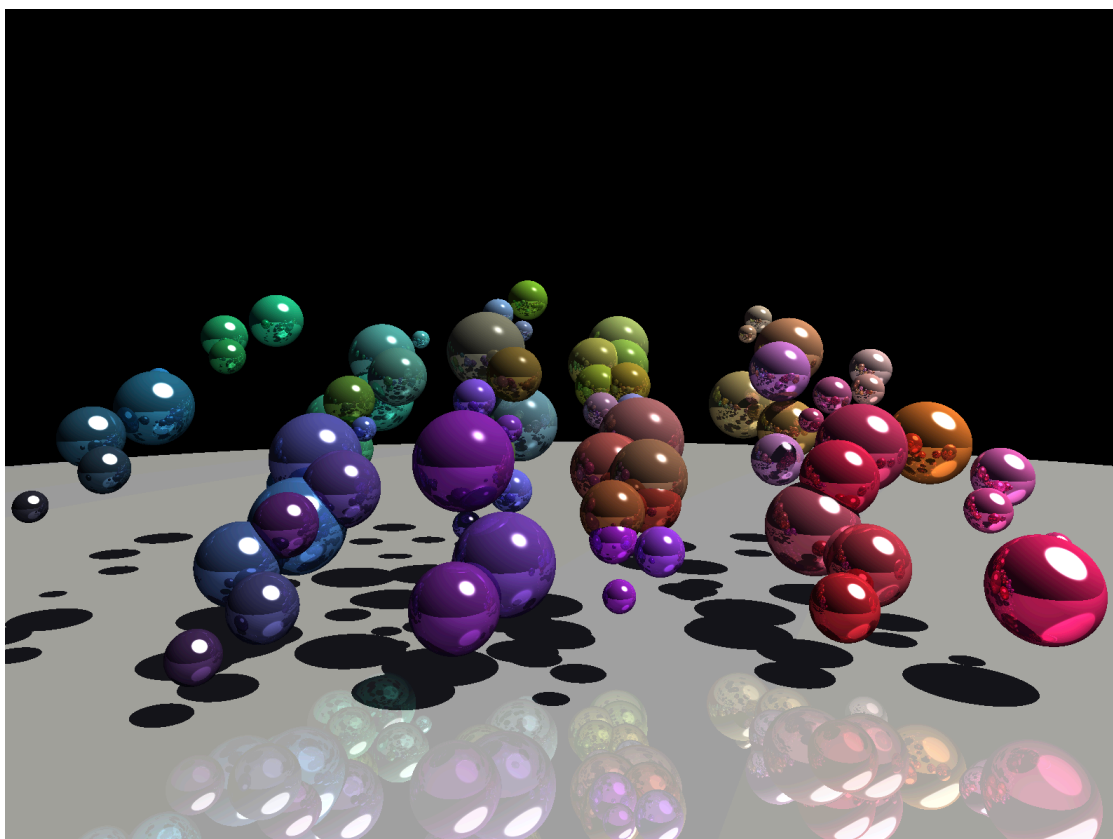
Závěr

Cílem této práce bylo navrhnout a implementovat algoritmus pro techniku *raytracing* s důrazem na různé metody paralelizace a následně porovnat jejich efektivitu. V rámci vývoje byly vytvořeny tři verze rendereru: sekvenční, vektorizovaná v kombinaci vícevláknovým zpracováním (využitím *ARM Neon* a *TBB*) a masivně paralelní pro GPU (využitím *Vulkan API*).

Nejsložitější částí práce byla optimalizace SIMD varianty výpočtu. Vzhledem k povaze algoritmu, který obsahuje relativně hodně větvení bylo potřeba často využívat maskování. To v praxi znamenalo provádění mnoha výpočtů, které byly nakonec zahozeny, což proces značně zpomalovalo. Ve výsledku se mi sice podařilo čistě vektorizovanou verzi optimalizovat v některých případech až na patnáctinásobek rychlosti původní neoptimalizované verze, i přesto ale přínos vektorizace nebyl příliš velký. Celkově se domnívám, že se pro tuto konkrétní úlohu vzhledem k její povaze příliš nehodí.

V porovnání všech přístupů dominovala verze pro GPU, která dosahovala pro 100 objektů skoro tisícinásobného zrychlení. Tato varianta se na rozdíl od vektorizace ukázala jako ideální pro paralelizaci této úlohy, což potvrzují i velmi nízké hodnoty Karp-Flattovy metriky.

Za velký osobní přínos považuji možnost detailně se seznámit s *Vulkan API*, protože toto rozhraní považuji za budoucnost vývoje GPU kódu. Vzhledem k dosaženým výsledkům a funkčnosti všech plánovaných rendererů považuji stanovený cíl práce za splněný.



Ukázka vyrenderované scény

Zdroje

[1] By Henrik - Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=3869326>

[2] “Apple M1,” *Wikipedia*. https://en.wikipedia.org/wiki/Apple_M1